Erstelle einen KI-gestützten Story-Bot

Dieser Blogbeitrag wurde mit Unterstützung von ChatGPT-4 verfasst.

Inhaltsverzeichnis

Einführung

Dieser Blogbeitrag bietet einen umfassenden Leitfaden zur Architektur und Implementierung einer KI-gestützten Story-Bot-Anwendung. Das Projekt beinhaltet die Generierung personalisierter Geschichten über eine Web-Oberfläche. Für die Entwicklung verwenden wir Python,

Flask und React und setzen die Anwendung auf AWS ein. Zusätzlich nutzen wir Prometheus für die Überwachung sowie ElasticSearch, Kibana und Logstash für das Log-Management. Das DNS-Management wird über GoDaddy und Cloudflare abgewickelt, während Nginx als Gateway für das SSL-Zertifikat und die Verwaltung von Anfrage-Headern dient.

Projektarchitektur

Backend Das Backend des Projekts wurde mit Flask erstellt, einem leichtgewichtigen WSGI-Webanwendungsframework in Python. Das Backend verarbeitet API-Anfragen, verwaltet die Datenbank, protokolliert Anwendungsaktivitäten und integriert Prometheus zur Überwachung.

Hier ist eine Aufschlüsselung der Backend-Komponenten:

1. Flask-Anwendungseinrichtung:
    - Die Flask-App wird initialisiert und so konfiguriert, dass sie verschiedene Erweiterungen wie Flask-CORS für die Handhabung von Cross-Origin Resource Sharing und Flask-Migrate für die Verwaltung von Datenbankmigrationen verwendet.
    - Die Anwendungsrouten werden initialisiert, und CORS wird aktiviert, um Cross-Origin-Anfragen zu ermöglichen.
    - Die Datenbank wird mit Standardkonfigurationen initialisiert, und ein benutzerdefinierter Logger wird eingerichtet, um Logeinträge für Logstash zu formatieren.

```
    from flask import Flask
    from flask_cors import CORS
    from .routes import initialize_routes
    from .models import db, insert_default_config
    from flask_migrate import Migrate
    import logging
    from logging.handlers import RotatingFileHandler
    from prometheus_client import Counter, generate_latest, Gauge

app = Flask(__name__)
app.config.from_object('api.config.BaseConfig')

db.init_app(app)
initialize_routes(app)
CORS(app)
migrate = Migrate(app, db)
```

2. Protokollierung und Überwachung:

    - Die Anwendung verwendet RotatingFileHandler, um Logdateien zu verwalten und formatiert die Protokolle mit einem benutzerdefinierten Formatter.
    - Prometheus-Metriken sind in die Anwendung integriert, um die Anzahl der Anfragen und die Latenz zu verfolgen.

```
    REQUEST_COUNT = Counter('flask_app_request_count', 'Gesamtzahl der Anfragen der Flask-App', ['metho
    REQUEST_LATENCY = Gauge('flask_app_request_latency_seconds', 'Anfragelatenz', ['method', 'endpoint'

def setup_loggers():
    logstash_handler = RotatingFileHandler('app.log', maxBytes=100000000, backupCount=1)
    logstash_handler.setLevel(logging.DEBUG)
    logstash_formatter = CustomLogstashFormatter()
    logstash_handler.setFormatter(logstash_formatter)

        root_logger = logging.getLogger()
        root_logger.setLevel(logging.DEBUG)
        root_logger.addHandler(logstash_handler)

app.logger.addHandler(logstash_handler)
werkzeug_logger = logging.getLogger('werkzeug')
werkzeug_logger.setLevel(logging.DEBUG)
werkzeug_logger.addHandler(logstash_handler)

setup_loggers()
```

3. Anfragebehandlung:

    - Die Anwendung erfasst Metriken vor und nach jeder Anfrage und generiert eine Trace-ID, um den Anfragefluss zu verfolgen.

```
    def generate_trace_id(length=4):
        characters = string.ascii_letters + string.digits
        return ''.join(random.choice(characters) for _ in range(length))

@app.before_request
def before_request():
    request.start_time = time.time()
```

```
    trace_id = request.headers.get('X-Trace-Id', generate_trace_id())
    g.trace_id = trace_id

    @app.after_request
    def after_request(response):
        response.headers['X-Trace-Id'] = g.trace_id
        request_latency = time.time() - getattr(request, 'start_time', time.time())
        REQUEST_COUNT.labels(method=request.method, endpoint=request.path, http_status=response.status_
        REQUEST_LATENCY.labels(method=request.method, endpoint=request.path).set(request_latency)
        return response
```

Frontend Das Frontend des Projekts wurde mit React erstellt, einer JavaScript-Bibliothek zur Entwicklung von Benutzeroberflächen. Es interagiert mit der Backend-API, um Story-Prompts zu verwalten, und bietet eine interaktive Benutzeroberfläche zur Erstellung und Verwaltung personalisierter Geschichten.

1. React-Komponenten:
    - Die Hauptkomponente verwaltet die Benutzereingaben für Story-Prompts und interagiert mit der Backend-API, um diese Geschichten zu verwalten.

```
    import React, { useState, useEffect } from 'react';
    import { ToastContainer, toast } from 'react-toastify';
    import 'react-toastify/dist/ReactToastify.css';
    import { apiFetch } from './api';
    import './App.css';
```

    function App() { const [prompts, setPrompts] = useState([]); const [newPrompt, setNewPrompt] = useState(’ ’); const [isLoading, setIsLoading] = useState(false);

```
useEffect(() => {
  fetchPrompts();
}, []);

  const fetchPrompts = async () => {
    setIsLoading(true);
    try {
      const response = await apiFetch('prompts');
      if (response.ok) {
```

```
      const data = await response.json();
      setPrompts(data);
    } else {
      toast.error('Fehler beim Abrufen der Prompts');
    }
  } catch (error) {
    toast.error('Ein Fehler ist beim Abrufen der Prompts aufgetreten');
  } finally {
    setIsLoading(false);
  }
};

const addPrompt = async () => {
  if (!newPrompt) {
    toast.warn('Der Prompt-Inhalt darf nicht leer sein');
    return;
  }
  setIsLoading(true);
  try {
    const response = await apiFetch('prompts', {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
      },
      body: JSON.stringify({ content: newPrompt }),
    });
    if (response.ok) {
      fetchPrompts();
      setNewPrompt('');
      toast.success('Prompt erfolgreich hinzugefügt');
    } else {
      toast.error('Fehler beim Hinzufügen des Prompts');
    }
  } catch (error) {
    toast.error('Ein Fehler ist beim Hinzufügen des Prompts aufgetreten');
  } finally {
```

```
    setIsLoading(false);
  }
};

const deletePrompt = async (promptId) => {
  setIsLoading(true);
  try {
    const response = await apiFetch(`prompts/${promptId}`, {
      method: 'DELETE',
    });
    if (response.ok) {
      fetchPrompts();
      toast.success('Prompt erfolgreich gelöscht');
    } else {
      toast.error('Löschen des Prompts fehlgeschlagen');
    }
  } catch (error) {
    toast.error('Ein Fehler ist beim Löschen des Prompts aufgetreten');
  } finally {
    setIsLoading(false);
  }
};

return (
  <div className="app">
    <h1>KI-gestützter Story-Bot</h1>
    <div>
      <input
        type="text"
        value={newPrompt}
        onChange={(e) => setNewPrompt(e.target.value)}
        placeholder="Neuer Prompt"
      />
      <button onClick={addPrompt} disabled={isLoading}>Prompt hinzufügen</button>
    </div>
    {isLoading ? (
```

```
        <p>Lade...</p>
      ) : (
        <ul>
          {prompts.map((prompt) => (
            <li key={prompt.id}>
              {prompt.content}
              <button onClick={() => deletePrompt(prompt.id)}>Löschen</button>
            </li>
          ))}
        </ul>
      )}
      <ToastContainer />
    </div>
  );
}

export default App;
```

2. API-Integration:
    - Das Frontend interagiert mit dem Backend-API über Fetch-Requests, um Story-Prompts zu verwalten.

```
export const apiFetch = (endpoint, options) => {
  return fetch(`https://api.yourdomain.com/${endpoint}`, options);
};
```

Bereitstellung

Das Projekt ist auf AWS bereitgestellt, wobei das DNS-Management über GoDaddy und Cloudflare abgewickelt wird. Nginx wird als Gateway für das SSL-Zertifikat und die Verwaltung der Anfrageheader verwendet. Für die Überwachung nutzen wir Prometheus, und für die Protokollverwaltung setzen wir ElasticSearch, Kibana und Logstash ein.

1. Deployment-Skript:
    - Wir verwenden Fabric, um Bereitstellungsaufgaben wie die Vorbereitung lokaler und entfernter Verzeichnisse, das Synchronisieren von Dateien und das Setzen von

Berechtigungen zu automatisieren.

```
from fabric import task
from fabric import Connection
```

server_dir = '/home/project/server' web_tmp_dir = '/home/project/server/tmp'

```
    @task
    def prepare_remote_dirs(c):
        if not c.run(f'test -d {server_dir}', warn=True).ok:
            c.sudo(f'mkdir -p {server_dir}')
            c.sudo(f'chmod -R 755 {server_dir}')
            c.sudo(f'chmod -R 777 {web_tmp_dir}')
            c.sudo(f'chown -R ec2-user:ec2-user {server_dir}')

@task
def deploy(c, install='false'):
    prepare_remote_dirs(c)
    pem_file = './aws-keypair.pem'
    rsync_command = (f'rsync -avz --exclude="api/db.sqlite3" '
                     f'-e "ssh -i {pem_file}" --rsync-path="sudo rsync" '
                     f'{tmp_dir}/ {c.user}@{c.host}:{server_dir}')
    c.local(rsync_command)
    c.sudo(f'chown -R ec2-user:ec2-user {server_dir}')
```

2. ElasticSearch-Konfiguration:

   - Die ElasticSearch-Einrichtung umfasst Konfigurationen für den Cluster, den Knoten und die Netzwerkeinstellungen.

```
cluster.name: my-application
node.name: node-1
path.data: /var/lib/elasticsearch
path.logs: /var/log/elasticsearch
network.host: 0.0.0.0
http.port: 9200
discovery.seed_hosts: ["127.0.0.1"]
cluster.initial_master_nodes: ["node-1"]
```

3. Kibana-Konfiguration:

    - Die Kibana-Einrichtung umfasst Konfigurationen für den Server und die ElasticSearch-Hosts.

```
server.port: 5601
server.host: "0.0.0.0"
elasticsearch.hosts: ["http://localhost:9200"]
```

4. Logstash-Konfiguration:

    - Logstash ist so konfiguriert, dass es Logdateien liest, sie analysiert und die analysierten Logs an ElasticSearch weiterleitet.

```
input {
  file {
    path => "/home/project/server/app.log"
    start_position => "beginning"
    sincedb_path => "/dev/null"
  }
}
```

filter { json { source => “message” } }

output { elasticsearch { hosts => [“http://localhost:9200”] index => “flask-logs-%{+YYYY.MM.dd}” } } “‘

Nginx-Konfiguration und Let’s Encrypt SSL-Zertifikat

Um eine sichere Kommunikation zu gewährleisten, verwenden wir Nginx als Reverse-Proxy und Let’s Encrypt für SSL-Zertifikate. Im Folgenden finden Sie die Nginx-Konfiguration für die Umleitung von HTTP zu HTTPS und die Einrichtung der SSL-Zertifikate.

1. Definiere eine Map, um die erlaubten Ursprünge zu verwalten:

```
map $http_origin $cors_origin {
    default "https://example.com";
    "http://localhost:3000" "http://localhost:3000";
    "https://example.com" "https://example.com";
    "https://www.example.com" "https://www.example.com";
}
```

2. HTTP auf HTTPS umleiten:

```
server {
    listen 80;
    server_name example.com api.example.com;

 return 301 https://$host$request_uri;

} "'
```

3. Hauptseitenkonfiguration für `example.com`:

"'nginx server { listen 443 ssl; server_name example.com;

```
 ssl_certificate /etc/letsencrypt/live/example.com/fullchain.pem;
 ssl_certificate_key /etc/letsencrypt/live/example.com/privkey.pem;

    ssl_protocols TLSv1.2 TLSv1.3;
    ssl_prefer_server_ciphers on;
    ssl_ciphers "EECDH+AESGCM:EDH+AESGCM:AES256+EECDH:AES256+EDH";

root /home/projekt/web;
index index.html index.htm index.php default.html default.htm default.php;

location / {
    try_files $uri $uri/ =404;
}

location ~ .*\.(gif|jpg|jpeg|png|bmp|swf)$ {
    expires 30d;
}

location ~ .*\.(js|css)?$ {
    expires 12h;
}

error_page 404 /index.html;
}
```

4. API-Konfiguration für `api.example.com`:

```
   server {
       listen 443 ssl;
       server_name api.example.com;

    ssl_certificate /etc/letsencrypt/live/example.com-0001/fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/example.com-0001/privkey.pem;

      ssl_protocols TLSv1.2 TLSv1.3;
      ssl_prefer_server_ciphers on;
      ssl_ciphers "EECDH+AESGCM:EDH+AESGCM:AES256+EECDH:AES256+EDH";

   location / {
       # Alle vorhandenen Access-Control-Header löschen
       more_clear_headers 'Access-Control-Allow-Origin';

       # CORS-Preflight-Anfragen behandeln
       if ($request_method = 'OPTIONS') {
           add_header 'Access-Control-Allow-Origin' $cors_origin;
           add_header 'Access-Control-Allow-Methods' 'GET, POST, OPTIONS, PUT, DELETE';
           add_header 'Access-Control-Allow-Headers' 'Origin, Content-Type, Accept, Authorization, X-C
           add_header 'Access-Control-Max-Age' 3600;
           return 204;
       }

       add_header 'Access-Control-Allow-Origin' $cors_origin always;
       add_header 'Access-Control-Allow-Methods' 'GET, POST, OPTIONS, PUT, DELETE' always;
       add_header 'Access-Control-Allow-Headers' 'Origin, Content-Type, Accept, Authorization, X-Clien

nginx             proxy_pass http://127.0.0.1:5000/;             proxy_set_header Host
$host;             proxy_set_header X-Real-IP $remote_addr;             proxy_set_header
X-Forwarded-For $proxy_add_x_forwarded_for;             proxy_set_header X-Forwarded-Proto
$scheme;             proxy_connect_timeout 600s;             proxy_send_timeout 600s;
proxy_read_timeout 600s;             send_timeout 600s;         }     }
```

Fazit

Dieses Projekt präsentiert eine robuste Architektur für eine KI-gestützte Story-Bot-Anwendung, die moderne Webentwicklungspraktiken und -tools nutzt. Das Backend wurde mit Flask entwickelt, um eine effiziente Anfrageverarbeitung und Integration mit verschiedenen Diensten für Protokollierung und Überwachung zu gewährleisten. Das Frontend, das mit React erstellt wurde, bietet eine interaktive Benutzeroberfläche zur Verwaltung von Story-Prompts. Durch die Nutzung von AWS für die Bereitstellung, Nginx für sichere Kommunikation und den ELK-Stack für die Protokollverwaltung stellen wir Skalierbarkeit, Zuverlässigkeit und Wartbarkeit sicher. Dieses umfassende Setup demonstriert die Kraft der Kombination modernster Technologien, um ein nahtloses Benutzererlebnis zu liefern.